Parsing Log Files

Welcome to Parsing Log Files on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

Package regexp offers support for regular expressions in Go.

Syntax

The syntax of the regular expressions accepted is the same general syntax used by Perl, Python, and other languages.

Both the search patterns and the input texts are interpreted as UTF-8.

When using backticks (`) to make strings, backslashes ( ) don't have any special meaning and don't mark the beginning of special characters like tabs `\t` or newlines `\n`:

```
"\t\n" // regular string literal with 2 characters: a tab and a newline
`\t\n`// raw string literal with 4 characters: two backslashes, a 't', and an 'n'
```

Because of this, using backticks is desirable to make regular expressions, because it means we don't need to escape backslashes:

```
"\\" // string with a single backslash
`\\` // string with 2 backslashes
```

Compiling patterns - `RegExp` type

To use a regular expression, we first must compile the string pattern. Compilation here means taking the string pattern of the regular expression and converting it into an internal

representation that is easier to work with. We only need to compile each pattern once, after that we can use the compiled version of the regular expression many times. The type `regexp.Regexp` represents a compiled regular expressions. We can compile a string pattern into a `regexp.Regexp` using the function `regexp.Compile`. This function returns `nil` and an error if compilation failed:

```
re, err := regexp.Compile(`(a|b)+`)
fmt.Println(re, err) // => (a|b)+ <nil>
re, err = regexp.Compile(`a|b)+`)
fmt.Println(re, err) // => <nil> error parsing regexp: unexpected ): `a|b)+`
```

Function `MustCompile` is a convenient alternative to `Compile`:

```
re = regexp.MustCompile(`[a-z]+\d*`)
```

Using this function, there is no need to handle an error.

```
`MustCompile` should only be used when we know for sure the pattern does compile, as oth
```

Regular expression methods

There are 16 methods of `Regexp` that match a regular expression and identify the matched text. Their names are matched by this regular expression:

```
Find(All)?(String)?(Submatch)?(Index)?
```

- If `All` is present, the routine matches successive non-overlapping matches of the entire expressions.
- If `String` is present, the argument is a string; otherwise it is a slice of bytes; return values are adjusted as appropriate.
- If `Submatch` is present, the return value is a slice identifying the successive submatches of the expression.
- If `Index` is present, matches and submatches are identified by byte index pairs within the input string.

There are also methods for:

- replacing matches of regular expressions with replacement strings and
- splitting of strings separated by regular expressions.

All-in-all, the `regexp` package defines more than 40 functions and methods. We will demonstrate the use of a few methods below. Please see the API documentation for details of these and other functions.

`MatchString` Examples

Method `MatchString` reports whether a string contains any match of a regular expression.

```
re = regexp.MustCompile(`[a-z]+\d*`)
b = re.MatchString("[a12]")       // => true
b = re.MatchString("12abc34(ef)") // => true
b = re.MatchString(" abc!")       // => true
b = re.MatchString("123 456")     // => false
```

`FindString` Examples

Method `FindString` returns a string holding the text of the leftmost match of the regular expression.

```
re = regexp.MustCompile(`[a-z]+\d*`)
s = re.FindString("[a12]")       // => "a12"
s = re.FindString("12abc34(ef)") // => "abc34"
s = re.FindString(" abc!")       // => "abc"
s = re.FindString("123 456")     // => ""
```

`FindStringSubmatch` Examples

Method `FindStringSubmatch` returns a slice of strings holding the text of the leftmost match of the regular expression and the matches, if any, of its subexpressions. This can be used to identify the strings matching capturing groups. A return value of `nil` indicates no match.

```
re = regexp.MustCompile(`[a-z]+(\d*)`)
sl = re.FindStringSubmatch("[a12]")       // => []string{"a12","12"}
sl = re.FindStringSubmatch("12abc34(ef)") // => []string{"abc34","34"}
sl = re.FindStringSubmatch(" abc!")       // => []string{"abc",""}
sl = re.FindStringSubmatch("123 456")     // => <nil>
```

`ReplaceAllString` Examples

Method `re.ReplaceAllString(src,repl)` returns a copy of `src`, replacing matches of the regular expression `re` with the replacement string `repl`.

```
re = regexp.MustCompile(`[a-z]+\d*`)
s = re.ReplaceAllString("[a12]", "X")       // => "[X]"
s = re.ReplaceAllString("12abc34(ef)", "X") // => "12X(X)"
s = re.ReplaceAllString(" abc!", "X")       // => " X!"
s = re.ReplaceAllString("123 456", "X")     // => "123 456"
```

`Split` Examples

Method `re.Split(s,n)` slices a text `s` into substrings separated by the expression and returns a slice of the substrings between those expression matches. The count `n` determines the maximal number of substrings to return. If `n<0`, the method returns all substrings.

```
re = regexp.MustCompile(`[a-z]+\d*`)
sl = re.Split("[a12]", -1)      // => []string{"[","]"}
sl = re.Split("12abc34(ef)", 2) // => []string{"12","(ef)"}
sl = re.Split(" abc!", -1)      // => []string{" ","!"}
sl = re.Split("123 456", -1)    // => []string{"123 456"}
```

Instructions

This exercise addresses the parsing of log files.

After a recent security review you have been asked to clean up the organization's archived log files.

All strings passed to the functions are guaranteed to be non-null and without leading and trailing spaces.

1. Identify garbled log lines

You need some idea of how many log lines in your archive do not comply with current standards. You believe that a simple test reveals whether a log line is valid. To be considered valid a line should begin with one of the following strings:

- [TRC]
- [DBG]
- [INF]
- [WRN]
- [ERR]
- [FTL]

Implement the `IsValidLine` function to return `false` if a string is not valid otherwise `true` .

```
IsValidLine("[ERR] A good error here")
// => true
IsValidLine("Any old [ERR] text")
// => false
IsValidLine("[BOB] Any old text")
// => false
```

2. Split the log line

A new team has joined the organization, and you find their log files are using a strange separator for "fields". Instead of something sensible like a colon ":" they use a string such as "<-->" or "<=>" (because it's prettier) in fact any string that has a first character of "<" and a last character of ">" and any combination of the following characters "~", "*", "=" and "-" in between.

Implement the `SplitLogLine` function that takes a line and returns an array of strings each of which contains a field.

```
SplitLogLine("section 1<*>section 2<~~~>section 3")
// => []string{"section 1", "section 2", "section 3"},
```

3. Count the number of lines containing `password` in quoted text

The team needs to know about references to passwords in quoted text so that they can be examined manually.

Implement the `CountQuotedPasswords` function to provide an indication of the likely scale of the manual exercise.

Identify log lines where the string "password", which may be in any combination of upper or lower case, is surrounded by quotation marks. You should account for the possibility of additional content between the quotation marks before and after "password". Each line will contain at most two quotation marks.

Lines passed to the routine may or may not be valid as defined in task 1. We process them in the same way, whether or not they are valid.

```
lines := []string{
    `[INF] passWord`, // contains 'password' but not surrounded by quotation marks
    `"passWord"`,  // count this one
    `[INF] User saw error message "Unexpected Error" on page load.`, // does not contain
    `[INF] The message "Please reset your password" was ignored by the user`, // count th
}
// => 2
```

4. Remove artifacts from log

You have found that some upstream processing of the logs has been scattering the text "end-of-line" followed by a line number (without an intervening space) throughout the logs.

Implement the `RemoveEndOfLineText` function to take a string and remove the end-of-line text and return a "clean" string.

Lines not containing end-of-line text should be returned unmodified.

Just remove the end of line string. Do not attempt to adjust the whitespaces.

```
RemoveEndOfLineText("[INF] end-of-line23033 Network Failure end-of-line27")
// => "[INF]  Network Failure "
```

5. Tag lines with user names

You have noticed that some of the log lines include sentences that refer to users. These sentences always contain the string `"User"`, followed by one or more space characters, and then a user name. You decide to tag such lines.

Implement a function `TagWithUserName` that processes log lines:

- Lines that do not contain the string `"User "` remain unchanged.
- For lines that contain the string `"User "`, prefix the line with `[USR]` followed by the user name.

For example:

```
result := TagWithUserName([]string{
    "[WRN] User James123 has exceeded storage space.",
        "[WRN] Host down. User   Michelle4 lost connection.",
        "[INF] Users can login again after 23:00.",
        "[DBG] We need to check that user names are at least 6 chars long.",

})
// => []string {
//  "[USR] James123 [WRN] User James123 has exceeded storage space.",
//  "[USR] Michelle4 [WRN] Host down. User   Michelle4 lost connection.",
//  "[INF] Users can login again after 23:00.",
//  "[DBG] We need to check that user names are at least 6 chars long."
// }
```

You can assume that:

- User names are followed by at least one whitespace character in the log.
- There is at most one occurrence of the string `"User "` in each line.
- User names are non-empty strings that do not contain whitespace.

Source

Created by

- @norbs57

Contributed to by

- @eklatzer